Energy Efficient and Reliable Maintenance for SDN-based Scheduled Wireless Networks

Farzad Veisi, Julien Montavont, Fabrice Theoleyre

Energy Efficient and Reliable Maintenance for SDN-based Scheduled Wireless Networks

Farzad Veisi, Julien Montavont and Fabrice Theoleyre

ICube, CNRS / University of Strasbourg
Bd Sébastien Brant, 67412 Illkirch, France
Emails: veisigoshtasb@unistra.fr, montavont@unistra.fr, and fabrice.theoleyre@cnrs.fr

***Abstract*—Wireless industrial networks represent a key enabler for Industry 4.0. Since wireless links are known to be lossy, most deployments rely on scheduled transmissions to avoid collisions. Software Defined Networking (SDN) is a famous architecture where a controller centralizes all the configuration. In a scheduled wireless network, the SDN paradigm has to be extended to allocate both paths and bandwidth. Unfortunately, most solutions address only the initial configuration, which becomes suboptimal when the link quality evolves, or the conditions change. We propose here the mechanisms for continuous optimization. We define how control packets can reconfigure the data plane while guaranteeing a globally consistent state, even when the control plane exploits unreliable links. No data packet is dropped because of inconsistent forwarding rules. To be energy efficient, we also propose a scheduling algorithm that minimizes the network reconfiguration to minimize the overhead. Our simulation results highlight the strength of our proposition to handle topology and link quality changes.**

***Keywords*—Industrial Internet of Things; Software Defined Networking; Scheduling; Resource allocation; Flow isolation**

I. Introduction

The Industrial Internet of Things (IIoT) [1] is considered a key enabler for Industry 4.0. Actuators and sensors may use wireless communications in smart factories and/ warehouses. Most deployments rely on scheduled wireless networks to provide low latency and high reliability. Thus, most IIoT technologies such as IEEE802.15.4-Time Slotted Channel Hopping (TSCH) [2] exploit scheduled access to the medium: each radio link can be activated during specific time windows on a predetermined radio frequency. The schedule is cyclic such that each active radio link receives a given bandwidth.

Software Defined Networking (SDN) [3] is very popular to separate the control and data planes to make the network easier to maintain and optimize. A controller exploits the so-called southbound API (*e.g.*, OpenFlow) to install rules in the forwarding devices. A network device follows simple rules defined in its flow table to forward any packet. Unfortunately, wireless networks create specific challenges for SDN. Indeed, wireless links are known to be lossy, making the control and data planes unreliable. Thus, a controller has to estimate the link quality to compute appropriate forwarding rules. Typically, SDNWISE-TSCH [4] defines a southbound API (*i.e.*, OpenPath packet) to install forwarding rules all along a path.

Forwarding rules are not enough for wireless networks. SDN-WISE [5] proposes additional mechanisms to implement also in-network packet processing. However, scheduled wireless networks need to schedule transmissions to work properly. In particular, unreliable links should have more retransmission opportunities to avoid packet drops. Unfortuantely, most distributed scheduling solutions have not been designed to provide guarantees [6].

Thus, we are convinced that the SDN controller must regroup two functions to support scheduled wireless networks:

forwarding rules are computed by the controller, and individual forwarding rules are pushed to each node;

resource is allocated by the controller. More precisely, the controller computes timeslots to use for each hop, and pushes this configuration in the network.

SDN-TSCH [7] has been to cope in scheduled wireless networks, even with unreliable links. However, it does not support reconfigurations. If the link quality fluctuates, the network may fail to meet the Service Level Agreements (SLAs) requirements.

In this paper, we propose the mechanisms to maintain and update the Software Defined Wireless Industrial Network. The contributions of this paper are as follows:

1) we provide a method to allow a controller to detect failing links, based on the link quality reported by each node;
2) we propose mechanisms to update both the control and data planes, redirecting all the flows through a new subpath when required. The method is **energy efficient** since it relies on a few control packet to reconfigure the network, and **reliable** because we prevent any inconsistent state during the reconfiguration, even if control packets are dropped;
3) we detail a (re)scheduling algorithm, that minimizes the number of changes to push in the network to be **energy efficient**.

II. Background & Related Work

We present here background notions on IEEE 802.15.4-TSCH (Time Slotted Channel Hopping), and detail SDN solutions for scheduled networks.

A. IEEE 802.15.4-TSCH networks

IEEE 802.15.4-TSCH is an operational amendment of the IEEE 802.15.4 standard designed for low-power industrial

wireless networks. It employs Time Division Multiple Access (TDMA) and multi-channel frequency hopping for reliable and energy-efficient communication. Transmissions are organized in a slotframe, which is a matrix of cells (pair of timeslot and channel offset). Slotframe is repeated over time. Each node is assigned a list of TX (transmit) and RX (receive) cells to wake up for transmitting and receiving packets respectively.

IEEE 802.15.4-TSCH defines two types of cells:

dedicated cell is allocated to one transmitter to prevent collisions during transmission. These collision-free cells are used typically for *e.g.*, data traffic or critical control packets;

shared cell can be used by more than one transmitter, and potential collisions can arise. For collision-resolution, the transmitter waits for a random number of shared cells to retransmit the packet, if it doesn't receive a acknowledgement. These shared cells are used for best-effort and broadcast traffic.

IEEE 802.15.4-TSCH is compatible with both distributed and centralized algorithms [6]. In the distributed approach, each node executes an algorithm to dynamically modify its local schedule. Minimal Scheduling Function (MSF) [8] is a popular standard: distributed cells are locally negotiated on demand. Each pair of transmitter/receiver dynamically adapts the schedule in response to the volume of data traffic. MSF usually relies on IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [9] to compute routes distributively.

By contrast, a centralized scheduler [10] may concentrate all the decisions. However, most solutions overlook the fact that the controller needs a complete view of the topology to detect interference, provision enough resources for each flow, etc. Besides, network maintenance is also challenging since the wireless conditions are known to be time-variant. In this article, we present an SDN-based industrial wireless network with continuous maintenance.

B. SDN for scheduled wireless networks

The SDN paradigm has to be specifically adapted to cope with scheduled wireless networks. Miranda *et al.* [11] propose a centralized architecture for a Time Sensitive Network (TSN) mixing WiFi 6 and wired segments. However, they do not support wireless multihop topologies. SDNWISE [5] proposes an SDN architecture that enables in-network packet processing. But, SDNWISE relies only on forwarding rules. Thus, SDNWISE-TSCH [4] defines additional mechanisms to support scheduled transmissions, which has been later extended to support multi-radio gateways (aka border routers) [12]. However, the resource allocation process is fixed, and the control plane relies on shared cells prone to collisions. Maintaining a globally consistent network is challenging in these conditions.

Papageorgiou *et al.* [13] allocate timeslots in a TDMA-based Mobile Ad-Hoc Network for military applications. One slot is allocated to each network node, even if it does not support data traffic, so that it can join the controller. However, the focus was rather on supporting highly dynamic topologies.

Whisper [14] relies on a centralized controller to control RPL and 6TiSCH Operation Sublayer (6P) protocols, aiming to create a centralized management system with minimal modifications to the network stack. The controller manipulates RPL rank to induce parent changes, establishing new paths. It also injects fake 6P commands for time-frequency block allocation or deallocation between nodes. Unfortunately, Whisper cannot respect SLA guarantees.

REACT [15] proposes a scheduling policy called *gap-induced* to reduce the latency and energy cost of network reconfiguration in industrial WirelessHART networks, a precursor of TSCH. The policy introduces intentional gaps between transmissions of the same flow. If the scheduler needs to update the schedule, this flexibility avoids a complete reconfiguration. However, to push the new schedule, REACT uses individual *DELETE* and *ADD* command packets, modifying the schedule one by one on each device. Changes in link quality can impact multiple flows, leading to a storm of control packets that may disrupt convergence.

SDN-TSCH [7] propose an SDN architecture for critical industrial applications with a focus on fulfilling SLAs. Any new device has to join first the TSCH network (link-layer synchronization) and then the SDN-TSCH network by contacting a controller. For this purpose, the new node sends a `report` packet to the controller, forwarded hop-by-hop to the controller through the control plane. In return, the controller can send a `config` packet toward the new node, using source-routing. Each intermediary node can read the headers of the `config` packet to update its schedule as decided by the controller. `report` and `report` packets correspond to a SDN Southbound API designed for wireless topologies. SDN-TSCH exploits only dedicated cells in the control plane so that each node has a reliable collision-free end-to-end bidirectional path with the controller. Unfortunately, the controller cannot update a schedule once it has been installed.

III. Wireless SDN Network Reconfiguration

We propose mechanisms in SDN-TSCH [7] to support a real-time network reconfiguration whenever the conditions evolve. Our solution is:

energy efficient : we can readjust bandwidth while over-provisioning leads to energy inefficiency, while under-provisioning may result in SLA violations. Besides, a `config` packet can configure a whole path: one end-to-end packet for each forwarding node is not anymore required.

reliable: we prevent any inconsistent state when the network is reconfigured. The forwarding rules are consistent and no data packet is dropped in the data plane, even if a SDN control packet needs to be retransmitted, or is dropped.

Let us consider the scenario illustrated in Fig. 1. If the quality of the link $F \rightarrow B$ decreases from 95% to 40%, the network must be reconfigured. The controller needs to i) detect the *fault*, ii) reconfigure the control plane so that F has dedicated cells with its new next hop toward the border

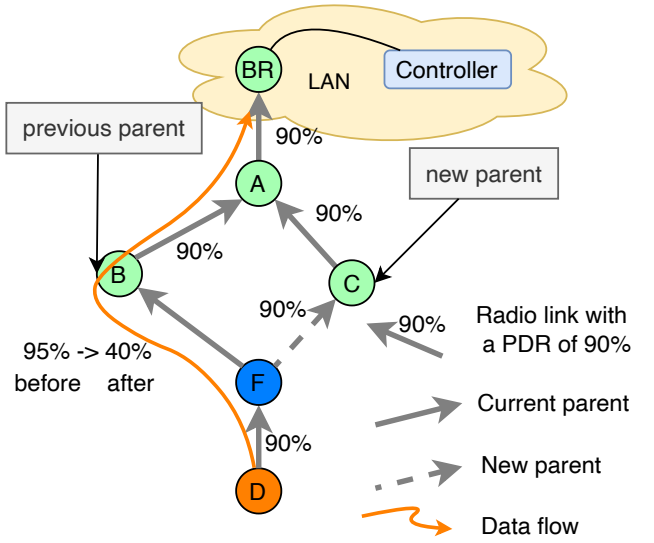

Fig. 1: A controller needs to reconfigure the control and data planes to redirect the flows through the link (F→C) when the PDR of the link (F→B) decreases from 95% to 40%

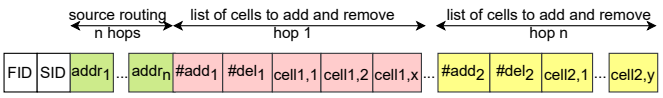

Fig. 2: Enhanced format of `config` packet

router (C), iii) redirect data flows (*e.g.*, $Flow_D$) by allocating resources in the data plane for the new path.

A. Fault detection & parent selection

Each IEEE 802.15.4-TSCH device sends periodically Enhanced Beacons (EBs) to announce its presence. EBs are used for synchronization, and to broadcast the network parameters. Moreover, in SDN-TSCH, each node sends periodically `report` packets to a controller, piggybacking the number of EBs received from its neighbors. This way, a controller is able to estimate the Packet Delivery Ratio (PDR) of each link by computing the ratio of received and expected EBs.

The controller needs to maintain a path with each node in the network. In particular, the link between any node with its parent must present a sufficient link quality to limit the number of retransmissions and packet drops. For each node, the controller compares continuously the Packet Delivery Ratio of its current parent, and the PDR of its best neighbor which could serve as new parent. A reconfiguration is engaged if the following condition holds:

$$PDR(current\_parent) \leq \alpha * PDR(best\_parent) \quad (1)$$

where $PDR()$ denotes the PDR value, $best\_parent$ is the neighbor with the largest PDR and which is not a descendant (no-loop condition), and $current\_parent$ is the current parent. $0 < \alpha < 1$ is the parameter defining the sensitivity of the reconfiguration rule (a larger value means more oscillations).

B. Control and data planes update

The controller needs first to reconfigure the control plane. More precisely, the path between the target node and the border router has to be updated to pass through the new parent. The controller has to allocate cells between the target node and its new parent (respectively F and C in Fig. 1). More precisely:

to controller: a new dedicated cell has to be reserved between the target node and its new parent. If the cell between the target node and its old parent is still free in the schedule of the new parent (half-duplex condition), the same cell is reused. Else, another cell is selected randomly to avoid collisions;

from controller: the new parent has already a transmitting cell to its children. Thus, the controller needs just to add the corresponding RX cell in the schedule of the target node.

The controller forges two `config` packets to push the new configuration, one per direction (to and from the controller). To route `config` packets to the target node, we utilize the same source-routing mechanism as the original SDN-TSCH version [7]. The `config` packet is forwarded through the dedicated cells in the `from controller` direction. In particular, the new parent will receive the packet to forward and will update its own schedule. For the last hop, a shared cell is used to reach the target node since no dedicated cell is present in the control plane for this specific (new) link.

To optimize the overhead of control packets, we enhance the `config` packet format defined in [7] to use the same packet to both install and remove cells (Fig. 2). Basically, the `config` packet piggybacks i) the route to follow (n nodes), ii) the list of cells to insert and to remove for each hop.

When the target node finally receives the `config` packet, it sends an end-to-end `ack` to the controller. It is worth noting that the target node stops using its previous parent as soon as the cells are installed in its schedule. After a timeout, the controller retransmits the `config` packet if no `ack` is received. This way, we maintain the global consistency of the schedule of the control plane.

When the target node has dedicated cells to and from its new parent for the control plane, the controller needs then to redirect the data flows through the new parent. Thus, it needs to send one `config` packet for each of the data flows. A `config` packet is forwarded through the control plane, which has already been reconfigured, using a similar method as before. The `config` packet follows the new path from the destination toward the source and exploits only dedicated cells to avoid collisions. As soon as the target node receives the `config` packet, data packets start to be transmitted through the new path for the corresponding data flow. It is worth noting that we guarantee flow isolation even during the convergence. We cannot have any inconsistent state since the path is configured up to the source node (upstream).

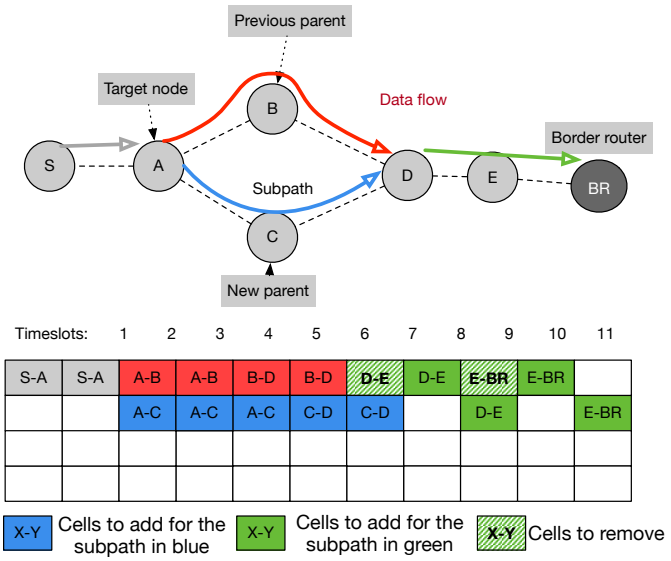

Fig. 3: Schedule update when redirecting a data flow

C. (Re)-scheduling algorithm

Computing a new schedule from scratch is expensive since we would have to remove all the previous cells and install a sequence of new cells all along the path. We propose rather a rescheduling algorithm that tries to minimize the number of schedule updates. Obviously, the problem is NP-complete, and we propose here a heuristic.

To minimize the changes, we focus on the subset of the path which differs between the previous and the new path. For instance, the subpath to update corresponds to the links $F \rightarrow C$ and $C \rightarrow A$ in Fig. 1. To reduce the end-to-end delay, we need to schedule the cells back-to-back, minimizing the buffering time. Moreover, it is more costly to change the schedule of nodes farther from the border router: the `config` packet has to be forwarded farther, possibly retransmitted, etc. Thus, we keep the same schedule from the source node to the target node and update only the rest of the schedule.

We adopt a greedy approach to compute a new schedule for the subpath to minimize the end-to-end delay [16]. First, the controller computes the number of cells to allocate for each hop, depending on the link quality. It greedily increases the number of cells on the weakest link in the subpath until the minimum end-to-end reliability is respected. Then, it selects free cells for each hop of the subpath. To minimize the forwarding delay, consecutive cells are selected preferentially (else, the closest cell is selected).

When the schedule of the subpath is computed, the controller may encounter two different cases:

valid: the last cell in the subpath is scheduled before the first cell of the rest of the path. In that case, the schedule is valid and is applied without modification;

overlap: the last cell of the subpath is scheduled after the first cell of the rest of the path. In that case, the controller updates the rest of the schedule hop-by-hop toward the border router. If the last cell of the current hop is scheduled **before** the cells for the next hop, the controller stops the updates. In the worst case, the schedule is updated up to the border router.

Algorithm 1: Re-scheduling of novel path

Data: $prev\_path\_num\_hops$, $prev\_schedule$, $prev\_path\_num\_cells\_hop$, $subpath\_num\_hops$, $subpath\_num\_cells\_hop$, $last\_cell\_of\_hop\_before\_subpath$, $first\_cell\_of\_hop\_after\_subpath$, $first\_hop\_index\_after\_subpath$
Result: $schedule$

```
1  lastCell ← last_cell_of_hop_before_subpath
2  for i = 1 to subpath_num_hops do      // schedule cells for subpath
3      for j = 1 to subpath_num_cells_hop[i] do
4          cell ← Pick first free cell after lastCell
5          schedule ← cell
6          lastCell ← cell
7  if lastCell > first_cell_of_hop_after_subpath then      // back-to-back condition is not met
8      for i = first_hop_index_after_subpath to prev_path_num_hops do
9          for j = 1 to prev_path_num_cells_hop[i] do
10             cell ← Pick first free cell after lastCell
11             schedule ← cell
12             lastCell ← cell
13             if lastCell < prev_schedule[i][j + 1] then
14                 PushSchedule(schedule) // Condition is met: stop scheduling
15 else
16     PushSchedule(schedule)      // Push the final schedule
```

For the sake of clarity, we described here only the case where the controller is the destination. However, this approach is obviously generalizable to any destination in the network.

Algorithm 1 presents the re-scheduling approach of the new path. Starting with the schedule of the subpath, it updates only the parts of the schedule that need to be changed to ensure that the end-to-end reliability and deadline are respected.

Let us consider the scenario illustrated in Fig. 3. The first two cells correspond to the path before the target node (A): they will not change. Then, the scheduler assigns consecutive cells for the new subpath (A,C,D) in blue. It is worth noting that the controller can assign the same timeslots through the old and new paths: it is sufficient to use different channel offsets to avoid collisions during the convergence.

This example shows an overlapping schedule: the last cell for the link $C \rightarrow D$ and the first cell of the link $D \rightarrow E$ are the same (timeslot 7). Thus, the controller has to assign hop-by-hop new cells (the cells in green) to update the rest of the schedule up to the border router. Symmetrically, the overlapping cells have to be deallocated (with diagonal

TABLE I: Simulation parameters

Simulation environment	OS: Contiki-ng (version 4.7) Simulator: Cooja https://github.com/Farzadv/Contiki-ng-SDN-TSCH.git Propagation model: Unit Disk Graph Medium Tx range = 100 m Interference range = 150 m Rx success = proportional to distance (100% - 0%) Number of runs per solution: 5
Application	Number of data flow: 1 flow per node Traffic pattern: Convergecast, Constant 1 packet every 5s Required QoS: PDR >= 99%, deadline <= 1500ms
SDN-TSCH	TSCH EB period: 15s SDN report period: 5 min `config` resend timeout: 50s Control plane slotframe duration: 2.5s (251 timeslots)

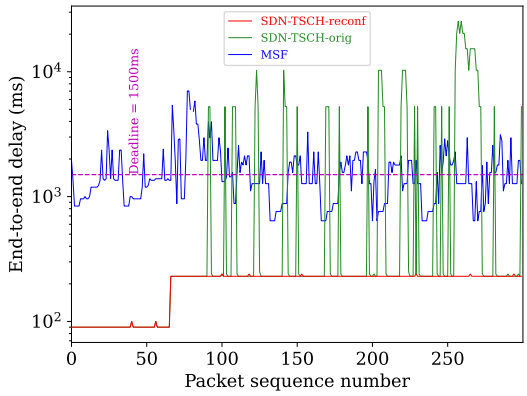

Fig. 4: Per-packet end-to-end delay of $Flow_F$ in each solution

hatching).

D. Obsolete cells removal

It is worth noting that some cells have been allocated and are not used anymore through the previous path. However, the energy consumption of unused cells is low [17]. Indeed, an unused TX cell has no cost: the transmitter does not wake up when no packet in its buffer has to be forwarded through the corresponding cell. Inversely, the receiver has to wake up, but it can turn its radio off after a fixed offset if no activity is measured on the medium.

We argue that the cost of unused cells is much smaller than sending and forwarding explicit `config` packets that have to be acknowledged end-to-end, retransmitted, etc. Thus, we propose a simple, timeout-based, schedule management. If no cell is used for a particular flow for a certain period of time, the corresponding cells are silently removed from the schedule. Each node maintains one timer per data flow, rearming the timer when it forwards a packet corresponding to the flow. Typically, the cells in red in Fig. 3 will be automatically removed from the schedule of B and D after the timeout.

IV. Performance Evaluation

We implement our solution in contiki-ng and Cooja and compare the following solutions:

SDN-TSCH-orig [7] is the original solution that does not update the schedule when a flow is configured;

SDN-TSCH-reconf is the proposition described in this paper, that detects when and which reconfiguration is required;

MSF [8] is a distributed scheduling function for TSCH[1].

We simulate the scenario illustrated in Fig. 1 representative of a topology with changing conditions. To more precisely analyze the impact of a reconfiguration, we consider a single link change. A real deployment may imply multiple reconfigurations by the controller but leading to the same conclusions. Each node hosts an application which requires an end-to-end PDR larger than 99% and an end-to-end delay lower than 1500ms. Table I regroups the values of the different parameters.

[1]https://github.com/alexrayne/contiki-ng.git

We measure first the end-to-end delay of $Flow_F$ (Fig. 4). The quality of the link $F \rightarrow B$ is manually changed when D generates the packet sequence number 90. SDN-TSCH-orig is able to deliver only some of the packets before the deadline: many of them require retransmissions or are dropped because of a buffer overflow. On the contrary, SDN-TSCH-reconf is able to reconfigure the network very efficiently, and all the packets are delivered before the deadline. It is worth noting that the delay increases after sequence number 90 because more retransmitting cells are required to compensate for the slightly lower PDR offered by the new path. Finally, MSF cannot provide any delay guarantee since it has not been designed for this purpose.

Fig. 5a reports the PDR of two data flows ($Flow_F$ and $Flow_D$) that have to be redirected through the link ($F \rightarrow C$). When the link quality degrades, SDN-TSCH-orig is not able to reconfigure the network, causing the end-to-end PDR to fall below the expected 99% level. The number of cells is not sufficient to cope with the required number of retransmissions. By contrast, SDN-TSCH-reconf is able to update the schedule, and it keeps on providing ultra high-reliability after the topology change. It is worth noting that MSF is not able to provision enough resources quickly, since many oscillations may arise. Thus, it provides a very low PDR after the quality of the link $F \rightarrow B$ is reduced.

We finally measure the reconfiguration time (Fig. 5b) and the overhead during the reconfiguration (Fig. 5c). We define the reconfiguration time as the time it takes to initiate the reconfiguration of a node and its subtree to adapt to the new path and schedule. SDN-TSCH-reconf implements an average reconfiguration of 14s. Indeed, one `config` packet is forwarded to or from the controller in one slotframe, which lasts 2.5 seconds. Since the control and data planes have to be reconfigured for two data flows (4 `config` packets), we need consequently several slotframes (at least one slotframe per `config` packet if we consider possible retransmissions). MSF needs on average 12 packets to readjust the bandwidth for the incriminated links, which remains very reasonable However, we need also to wait for the convergence

(a) Per-flow PDR (b) Reconfiguration time (c) Control overhead for reconfiguration

Fig. 5: Comparison of SDN-TSCH-orig, SDN-TSCH-reconf, and MSF

of RPL to have consistent routing tables and then update the local schedules. Possibly, RPL may experience temporary oscillations during such a convergence. Thus, MSF converges much slower than SDN-TSCH-reconf.

In addition to the reconfiguration time, the network should also detect when a significant change occurs. Because of the lack of space, we provide here only numerical values. MSF experiences very long detection times, ranging from 1.5 to 18 minutes. Considering SDN-TSCH-reconf, a reconfiguration is triggered upon reception of a `report` packet. In the worst case, the change occurs just after F sends its `report`, so the detection time is bounded by the `report` period set to 5min in our simulations.

V. Conclusion & Perspectives

We present here all the mechanisms to reconfigure a SDN scheduled wireless network. We reduce the overhead, with a single control packet to configure a whole path, and present how we can prevent any inconsistent state, even temporarily. We also present a scheduling algorithm to minimize the amount of reconfiguration to make the maintenance energy efficient, and to minimize the impact of unreliability.

In future work, we plan to improve fault detection in the network by implementing advanced techniques. Machine learning techniques may help to predict network faults by analyzing real-time network monitoring data and historical performance metrics. Our controller-based architecture is promising since all the measurements are consolidated in a single location. Finally, we need also to reduce the detection time when a fault occurs, to notify immediately the controller.

Acknowledgment

This work was partly supported by the Region Grand Est.

References

[1] Y. Wu *et al.*, "A survey of intelligent network slicing management for industrial iot: Integrated approaches for smart transportation, smart energy, and smart factory," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 2, pp. 1175–1211, 2022.

[2] "IEEE Standard for Low-Rate Wireless Networks," IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015), 2020.

[3] N. Bizanis *et al.*, "SDN and virtualization solutions for the Internet of Things: A survey," *IEEE Access*, vol. 4, pp. 5591–5606, 2016.

[4] F. Orozco-Santos *et al.*, "Enhancing SDN WISE with Slicing Over TSCH," *Sensors*, vol. 21, no. 4, 2021.

[5] L. Galluccio *et al.*, "SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for Wireless Sensor networks," in *INFOCOM*, 2015.

[6] R. Teles Hermeto *et al.*, "Scheduling for IEEE802.15.4-TSCH and Slow Channel Hopping MAC in Low Power Industrial Wireless Networks," *Comput. Commun.*, vol. 114, no. C, pp. 84–105, Dec. 2017.

[7] F. Veisi *et al.*, "Enabling centralized scheduling using software defined networking in industrial wireless sensor networks," *IEEE Internet of Things Journal*, 2023.

[8] T. Chang *et al.*, "6TiSCH Minimal Scheduling Function (MSF)," IETF, RFC 9033, 2021.

[9] T. Winter, "Routing protocol for low-power and lossy networks," IETF, RFC 6550,6551,6552, 2012.

[10] N. Taheri Javan *et al.*, "IEEE 802.15.4.e TSCH-Based Scheduling for Throughput Optimization: A Combinatorial Multi-Armed Bandit Approach," *IEEE Sensors Journal*, vol. 20, no. 1, pp. 525–537, 2020.

[11] G. Miranda *et al.*, "Enabling time-sensitive network management over multi-domain wired/wi-fi networks," *IEEE TNSM*, 2023.

[12] F. Orozco-Santos *et al.*, "Scalability enhancement on software defined industrial wireless sensor networks over tsch," *IEEE Access*, vol. 10, pp. 107 137–107 151, 2022.

[13] Y. Papageorgiou *et al.*, "Joint controller placement and tdma link scheduling in sdn-enabled tactical manets," in *MILCOM*. IEEE, 2022, pp. 125–132.

[14] E. Municio *et al.*, "Whisper: Programmable and flexible control on industrial IoT networks," *Sensors*, vol. 18, no. 11, p. 4048, 2018.

[15] D. Gunatilaka *et al.*, "REACT: An agile control plane for industrial wireless sensor-actuator networks," in *IoTDI*. IEEE, 2020, pp. 53–65.

[16] G. Gaillard *et al.*, "Kausa: KPI-aware Scheduling Algorithm for Multi-flow in Multi-hop IoT Networks," in *ADHOC-NOW*, 2016.

[17] X. Vilajosana *et al.*, "A Realistic Energy Consumption Model for TSCH Networks," *IEEE Sensors Journal*, vol. 14, no. 2, pp. 482–489, 2014.